Attack Pattern Usage

Sean Barnum, Cigital, Inc. [vita[3]]
Amit Sethi, Cigital, Inc. [vita[4]]

Copyright © 2006 Cigital, Inc.

2006-11-07 L3 / L, M[5]

This article is the third in a coherent series introducing the concept, generation, and usage of attack patterns as a valuable knowledge tool in the design, development, and deployment of secure software. It is recommended that the reader review the preceding articles to fully understand the context of the material presented.

Unlike many other concepts and tools with a narrowly focused area of impact, attack patterns provide potential value during all phases of software development regardless of the SDLC chosen, including requirements, architecture, design, coding, testing, and even deploying the system. However, because attack patterns describe how an attacker may break software, some readers may not immediately understand how attack patterns can be used to actually build secure software. Once the reader grasps the importance of understanding the attacker's perspective to software security, the value of attack patterns becomes intuitively clear. Without knowing how software may be attacked, it is difficult to know how to defend against the attacks.

The sections below describe how attack patterns can be leveraged during each stage of the SDLC. To make the information more concrete, each section provides an example. All examples will use as their basis one application that needs to be developed. The application will be web based and designed to let consumers purchase books online.

Requirement Gathering

This article assumes that the reader is familiar with the basic activities and results of typical software requirements definition efforts. Many other resources explain the various methodologies and challenges for requirements gathering. The Build Security In site offers a good Requirements Engineering Annotated Bibliography[7]. Discussion here will focus on the role attack patterns play in defining more appropriate and comprehensive requirements regarding the security of the software under development.

Functional Requirements

Most requirements gathering starts with relatively high-level functional requirements such as "users shall be able to access the site using at least the latest versions of Internet Explorer and Mozilla Firefox" and "users shall be able to purchase books in any currency". These high-level requirements generally lead to more detailed functional requirements and can potentially drive out security requirements. These security requirements can be functional, whether visible to the end user or not, or not functional in nature, but equally important. Very often, detailed functional and non-functional requirements including security requirements are overlooked and neglected because the general focus is basic functionality.

Deriving Security Requirements From Functional Requirements

The above two requirements should lead to questions that could help identify security requirements. If a user attempts to view the website with anything but the latest versions of Internet Explorer and Mozilla Firefox, what should happen? Is it acceptable if the browser crashes? Is it acceptable if absolutely nothing is displayed? Is there anything that the server needs to do to differentiate between browsers? What should

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/35-BSI.html (Barnum, Sean)
4. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/601-BSI.html (Sethi, Amit)
7. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/requirements/231-BSI.html (Requirements Engineering Annotated Bibliography)

happen if the self-identification data sent by the client is spoofed (e.g., if Mozilla Firefox is set to report itself as being Internet Explorer)? Also, if users can purchase books in other currencies, then should they be able to browse the website in other languages or encoding schemes (e.g., Unicode)? If so, how many languages and encoding schemes should the website support? What should happen if a client sends characters from a language or encoding scheme that the server does not accept?

As shown above, the process of making functional requirements more specific often is also an effective mechanism for identifying security requirements. For instance, indicating that "if a client sends characters from a language that the server does not recognize, then the server will return a HTTP 415 status code" is a good security requirement. This informs the developers how to handle the issue. Otherwise, the problem may be overlooked, causing issues such as attackers being able to bypass input filters.

Positive and Negative Security Requirements: The Role of Attack Patterns

Security-focused requirements are typically further split between positive requirements, which specify functional behaviors the software must exhibit (often security features), and negative requirements (typically in the form of misuse/abuse cases), which describe behaviors that the software must not exhibit to be operating securely [McGraw 06[8]].

Attack patterns can be an invaluable resource for helping to identify both positive and negative security requirements. They have obvious direct benefit in defining the software's expected reaction to the attacks they describe. When put into the context of the other functional requirements for the software and when considering the underlying weaknesses targeted by the attack, they can help identify both negative requirements describing potential undesired behaviors and positive functional requirements for avoiding, or at least mitigating, the potential attack. For instance, if a customer provides the requirement "the application must accept ASCII characters," then the attack pattern "Unicode Encoding" can be used to ask the question "What should the application do if Unicode characters or another unacceptable character set is encountered?" From this question, misuse/abuse cases can be defined such as "Malicious user provides Unicode characters to the data entry field." By having a specific definition for this negative requirement, the designers, implementers, and testers will have a clear idea of the type of hostile environment with which the software must deal and will build the software accordingly. This information can also help define positive requirements such as "The system shall filter all input for Unicode characters." If these sorts of requirements are overlooked, the developed application may have instances in which it may unknowingly accept Unicode characters, and an attacker could use that fact to bypass input filters for ASCII characters.

Many vulnerabilities result from vague specifications and requirements. This includes ambiguities outside the immediate scope of the application, including "unspecified behavior" in certain specifications (e.g., C language and how compilers must deal with certain situations) or RFCs (e.g., IP fragmentation and how end nodes interpret the specification in varying fashions). Requirements should specifically address these ambiguities to avoid opening up multiple security holes. In general, attack patterns allow the requirements gatherer to ask "what if" questions to make the requirements more specific. If an attack pattern states "Condition X can be leveraged by an attacker to cause Y," then a valid question may be "What should the application do if it encounters condition X?"

Varying Levels of Attack Pattern Detail and Specificity

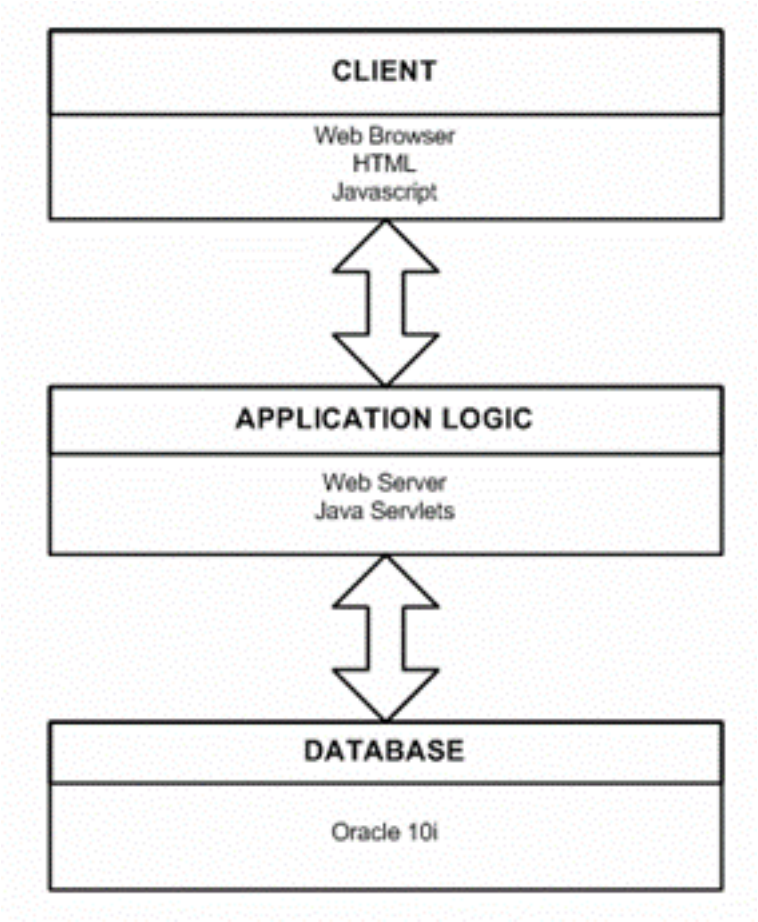
Attack patterns can exist at varying levels of detail and specificity; they often may start out more abstract with less known instances of exploit and then mature in level of detail over time as more exploit instances are discovered. These differing levels of detail also can influence the requirements they identify at different levels. More abstract attack patterns typically lead to less specific nonfunctional requirements, while more detailed attack patterns typically lead to more specific functional requirements.

8. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_mcgraw06 (Attack Pattern References)

Architecture and Design

Once requirements have been defined, all software must go through some level of architecture and design. Regardless of the formality of the process followed, the results of this activity will form the foundation for the software and drive all remaining development activities. During architecture and design, decisions must be made about how the software will be structured, how the various components will integrate and interact, which technologies will be leveraged, and how the requirements defining how the software will function will be interpreted. Careful consideration is necessary during this activity, as up to 50% of software defects leading to security problems are design flaws [McGraw 06[9]]. In the example in Figure 1, a potential architecture could consist of a three-tier system with the client (a web browser leveraging Javascript/ HTML)), a web server (leveraging Java™ Servlets), and a database server (leveraging Oracle 10i). Decisions made at this level can have a significant impact on the overall security profile of the software.

Figure 1. Example architecture



Attack patterns can be valuable during architecture and design in two ways. First, some attack patterns describe attacks that directly exploit architecture and design flaws in software. For instance, the “Make the Client Invisible” attack pattern described in the Introduction to [10]Attack Patterns[11] article exploits client-side trust issues that are apparent in the software architecture. Second, attack patterns at all levels can provide a useful context for the threats that the software is likely to face and thereby determine which architectural and design features to avoid or to specifically incorporate. The Make the Client Invisible attack pattern tells us that absolutely nothing sent back by the client can be trusted, regardless of what network security mechanisms (e.g., SSL) are used. The client is untrusted, and an attacker can send back literally any

9. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_mcgraw06 (Attack Pattern References)
10. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/585-BSI.html (Introduction to Attack Patterns)
11. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/585-BSI.html (Introduction to Attack Patterns)

information that he/she desires. All input validation, authorization checks, etc. must be performed on the server side. In addition, any data sent to the client should be considered visible by the client regardless of its intended presentation (i.e., data that the client should not see should never be sent to the client). Performing authorization checks on the client side to determine what data to display is unacceptable.

The Make the Client Invisible attack pattern instructs the architects and designers to ensure that absolutely no business logic is performed on the client side. In fact, depending on the system requirements, and the threats and risks the system faces, the architects and designers may even want to define an input validator through which all input to the server must pass before being sent to the other classes. Such decisions must be made at the architecture and design phase, and attack patterns provide some guidance regarding what issues should be considered.

It is essential to document any attack patterns used in the architecture/design phase so that the application can be tested using those attack patterns. Tests must be created in the later testing phase to validate that mitigations for the attack patterns considered during this phase were implemented properly.

Implementation and Coding

If architecture and design have been performed properly, each developer implementing the design should be writing well-defined components with well-defined interfaces.

Attack patterns can be useful during implementation because they identify the specific weaknesses targeted by relevant attacks and allow the developer to ensure that these weaknesses do not occur in their code. These weaknesses could take the form of implementation bugs or simply valid coding constructs that bear with them security implications. Implementation bugs are not always easy to avoid or to catch and fix. Even after applying basic review techniques, they can still remain abundant and can make software vulnerable to extremely dangerous exploits. It is important to extend basic review techniques with more focused security relevant concerns. Failure to properly check an array bound, for example, can lead to an attacker being able to execute arbitrary code on the target host. Failure to perform proper input validation can lead to an attacker being able to destroy an entire database. Underlying security issues in non-buggy valid code are typically more difficult to identify. They cannot be tested for with a functional behavioral model the way bugs can. They require specialized knowledge of what these weaknesses look like. These articles focus on how attack patterns can be used to identify specific weaknesses for targeting and mitigation through informing the developer ahead of time of the issues to avoid and through providing a list of issues (Security Coding Rules[13]) to look for in code reviews, often performed with security scanning tools.

Prevention requires that the developers understand applicable attack patterns and ensure that their code does not allow the attack patterns to succeed. The first step is to determine which attack patterns are applicable for the application being developed. Only a subset of attack patterns will be applicable for a particular piece of software, depending on its architecture, environment, and the technologies used to implement it. For instance, buffer overflow vulnerabilities are not typically applicable if all coding is done in Java. Input validation vulnerabilities may be less of a concern if all untrusted input is passed through a vetted, central, server-side filter before it is delivered to their code, rather than relying on all entry points (often implemented by different individuals) to perform their own validation. It is important to determine the attack patterns that will be applicable for a particular project. In some instances, different attack patterns may be applicable for different components of a product.

Once the applicable attack patterns are determined, they can be used to guide developers as to what not to allow in their code. In our example, a developer could leverage an attack pattern such as “simple script injection” and avoid XSS vulnerabilities. One relatively easy way to do this is to identify all places from which output is being sent to the user from an untrusted source and convert potentially dangerous characters into their HTML equivalents. For instance, convert ”<” to ”<”, ”>” to ”>”, etc. Third-party libraries for Java can perform such conversions automatically. JavaScript’s escape() function performs a similar task. This will prevent untrusted input containing potentially malicious data from being displayed to the

13. http://buildsecurityin.us-cert.gov/bsi/76-BSI.html (Coding Rules)

user. Malicious data could include artifacts such as <script> tags inserted by an attacker. This conversion should be carefully managed to avoid potential unintended buffer overflow issues. Of course, this problem could also be handled in other ways, such as use of a white list or at an architectural level by defining an input validator and an output sanitizer. The architectural approach would be more suitable for large projects, whereas dealing with the problem at the implementation level may be acceptable for smaller projects.

Good architecture/design as well as developer awareness, enhanced with attack patterns, can potentially help to minimize many security weaknesses. However, it is also essential to ensure that all source code, once written, is reviewed to validate the absence of targeted weaknesses. Due to the size and monotony of this task, it is typically performed using an automated analysis tool[14] (e.g., those from Fortify, Klocwork, Coverity). Even though analysis tools cannot find all security weaknesses, they can help weed out many potential issues. Using attack patterns as guidance, specific subsets of the tools' search rules[15] can be targeted and custom rules can be created for organizations to help find security weaknesses or instances of failure to follow security standards. For example, revisiting the potential "Simple Script Injection" attack pattern, an organization may have a security standard in which all untrusted input is passed through an input filter, and all output of data obtained from an untrusted source is passed through an encoder. An organization can develop such filters and encoders, and static source code analysis tools can help find occurrences in code where developers may have neglected to adhere to standards and opted to use Java's input/output features directly.

Software Testing and Quality Assurance

Testing and quality assurance[16] is a critical phase in the software development lifecycle. Software must undergo several levels and types of testing before it is released into a production environment. Different levels of testing include unit testing, integration testing, system testing, regression testing, and deployment testing. Different types of testing include functional testing, security testing (including penetration testing), performance testing, data integrity testing, and stress testing. A detailed discussion about all of the various levels and types of testing is out of scope for this paper. However, it is important to note that attack patterns can be leveraged during many different levels and types of testing to help design test cases.

The testing phase is different than the previous ones in the SDLC in that its goal is not necessarily constructive; the goal of risk-based security testing is typically to attempt to break software so that the discovered issues can be fixed before an attacker can find them [Whittaker 03[17]]. The purpose of using attack patterns in this phase is to have the individuals performing the various levels and types of testing act as attackers attempting to break the software.

Leveraging Attack Patterns in Unit Testing

Unit testing involves testing the components or pieces of software independently to ensure that they meet their functional and non-functional specifications. Applicable attack patterns should be used to identify relevant targeted weaknesses and to generate test cases for each component to ensure that they avoid or resist these weaknesses. For example, to test for shell command injection using command delimiters, malicious input strings containing delimiter separated shell commands should be crafted and input to the applicable component(s) to ensure proper behavior when provided with this type of malicious data.

Leveraging Attack Patterns in Integration Testing

Integration testing involves ensuring that software components integrate and interact together properly. This requires not only ensuring that all components compile together and that their interfaces match but also that the actual functionality of the components does not conflict If a good architecture and design are created and

14. http://buildsecurityin.us-cert.gov/bsi/articles/tools/code.html (Source Code Analysis)
15. http://buildsecurityin.us-cert.gov/bsi/33-BSI.html (Coding Rules Overview)
16. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing.html (Security Testing)
17. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_whittaker03 (Attack Pattern References)

proper unit testing is performed, integration testing should reveal less major issues than otherwise. A primary security issue to consider during integration testing is whether the individual components make differing assumptions based on security such that the integrated whole may contain conflicts or ambiguities. Attack patterns can be leveraged to create some test cases for integration testing. At a minimum, the attack patterns documented in the architecture/design phase should be used to create integration tests. Other attack patterns may be applicable as well. For instance, the Make the Client Invisible attack pattern can be used to create test cases that simulate an attacker bypassing the client and communicating directly with the server or an attacker modifying the client to send malicious data to the server.

Leveraging Attack Patterns in System Testing

System testing is used to test the entire system to ensure that it meets all of its functional and non-functional requirements. Hopefully, attack patterns were used in the requirement gathering phase to generate security requirements. These security requirements should be tested during system testing. For example, the Unicode Encoding attack pattern can be used to generate test cases that ensure that the application behaves properly when provided with unexpected characters. Testers should provide characters that the application is not supposed to accept to the application to see how it behaves. The application's actual behavior when under attack should be compared with the desired behavior defined in the security requirements.

Leveraging Attack Patterns in Regression Testing

Regression testing is the running of existing tests on the software any time that the code is changed to ensure that the change not only caused the intended behavior but also that it did not inadvertently cause any unintended changes. Attack patterns do not bring any new and unique value to regression testing itself. Effective regression testing should include security test cases developed during the other testing levels that were guided by use of attack patterns.

Leveraging Attack Patterns for Testing in the Operational Environment

Even after application of typical testing levels, software brings with it security concerns applicable to testing. Even if security was considered throughout the SDLC when building software, and even if extensive testing has been performed, vulnerabilities will likely still exist in the software. This is because no useful piece of software is 100 percent secure [Viega 01[18]]. For software to be useful, there must be ways to use it. Revisiting the analogy of a bank vault, a vault could be made extremely secure if it were constructed a few miles underground, was surrounded by several hundred feet of steel-reinforced concrete, had no access doors, and could withstand attacks from nuclear bombs. It might even be 100 percent secure, but it would of course be completely useless. For it to be useful, there must be a way to access it, and an attacker is likely to exploit the access point(s) if the access point(s) are the easiest ways of gaining access. Designers and builders can only ensure that they disallow "side-channel" attacks, so that the only way the attacker can access the vault is through the door built into it. The designers can do absolutely nothing to prevent an authorized employee from giving away the combination to the vault to their friends, from leaving the door ajar, etc. The vault itself can be made extremely secure, but the actual operational environment may be completely insecure. Software also faces similar issues. Software can be designed and developed to be extremely secure, but if it is deployed and operated in an insecure fashion many vulnerabilities can be introduced. For example, a piece of software could provide strong encryption and proper authentication before allowing access to encrypted data, but if an attacker can obtain valid authentication credentials he/she can subvert the software's security. Nothing is 100 percent secure, and the environment must be secured and monitored to thwart attacks.

Newer object-oriented programming models involving principles such as inversion of control further complicate the problem. For instance, the Spring framework for Java allows components to be "wired" together declaratively, similar to components being assembled together in a car. The entire car does not need to be rebuilt if a manufacturer decides to use a different brand of tires; the Spring framework enables similar

18. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_viega01 (Attack Pattern References)

swapping of software components during deployment without requiring rebuilding of large pieces. However, the problem is that the system designers and developers may have made assumptions regarding certain components that may not be satisfied by the components that are actually deployed. Such issues cannot be considered the manufacturer's fault unless they provide insufficient documentation.

Therefore, it is extremely important to perform security testing of the software in its actual operational environment. Vulnerabilities present in software can sometimes be masked by environmental protections such as network firewalls and application firewalls, and environmental conditions can sometimes create new vulnerabilities. Such issues can often be discovered using a mix of white-box and black-box analysis of the deployed environment. White-box analysis of deployed software involves performing security analysis of the software, including its deployed environment, with knowledge of the architecture, design, and implementation of the software. Black-box analysis (typically in the form of penetration testing) involves treating the deployed software as a "black box," and attempting to attack it without any knowledge of its inner workings. While black-box analysis is relatively inexpensive and can find many of the more obvious and small problems, it is not as effective at finding many of the often more significant issues. These issues are typically found only through in-depth white-box analysis. Black-box is good for finding the specific implementation issues you know to look for, while detailed and structured white-box can uncover unexpected architecture/design and implementation issues that you may not have known to look for. Both types of testing are important, and attack patterns can be leveraged for both.

Leveraging Attack Patterns for Black-Box Testing

Black-box testing[19] of web applications is generally performed using tools such as application security testers like those from companies such as SPI Dynamics[20] that automatically run predefined tests. Attack patterns can be used as models to create the tests these tools perform, thereby giving them more significant relevance and effectiveness. Such tools, though, cannot find many types of architectural flaws, or even all implementation errors. These tools generally test for a large variety of attacks, but they generally cannot find subtle architectural vulnerabilities. They effectively find issues that script kiddies and other relatively unskilled attackers would likely exploit. However, a skilled attacker would be able to find many issues that a vulnerability scanning tool simply could not detect. For instance, a lack of encryption for transmitting social security numbers would not be detected using an automated tool, as the fact that social security numbers are unencrypted is not a purely technical flaw. The black-box testing tool cannot determine what information is a social security number and cannot apply business logic. Attack patterns that are useful for creating black-box tests include those that can be executed remotely without requiring many steps. Some examples of vulnerabilities that black-box testing can detect include cross-site scripting using injection of JavaScript in a HTTP parameter and SQL injection using separator characters. Automated tools can be used to create tests, such as where a separator character is inserted into a HTML form field, to observe whether a database error occurs. Black-box testing of non-web applications can be performed similarly using different tools.

Leveraging Attack Patterns for White-Box Testing

White-box testing[21] is slower but more thorough than black-box. It involves extensive analysis performed by security experts that have access to the software's requirements, architecture, design, and code. The primary goal of white-box security testing is to find the more obscure implementation bugs not found in black-box testing as well as architecture and design flaws and related security issues. The advantage of white-box testing lies in its thoroughness; security experts may analyze a system for several weeks or months while knowing all of its internal details. If the flaws they find are mitigated, it is unlikely that an attacker with limited knowledge of an application's internal workings will easily find a significant vulnerability. Attack patterns can be leveraged to determine areas of system risk and thereby on which areas of the system white-box analysis should focus. The attack patterns most effective for white-box analysis include those that target architecture and design weaknesses. Attack patterns that target specific implementation weaknesses

19. http://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box.html (Black Box Testing)
20. http://www.spidynamics.com/
21. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box.html (White Box Testing)

should not be completely disregarded because, in many cases, implementation weaknesses can only be easily found using manual code reviews (a type of white-box analysis). An attack pattern that could be leveraged in white-box testing of a deployed system is sniffing sensitive data on an insecure channel. Those with knowledge of data sensitivity classifications and an understanding of the business context around various types of data can determine if some information that should always be communicated over an encrypted channel is actually sent over an insecure channel. Such issues are often specific to a deployed environment; thus, analysis of the actual deployed software is required.

Leveraging Attack Patterns in the Broader Spectrum of Testing

The testing phase of the SDLC is vital to ensuring the security of the software under development, and, as outlined here, attack patterns can play a valuable and broad role across the various testing activities. There are other more specific types of testing where attack patterns could be leveraged explicitly or implicitly to test the security of the system, but that level of detail exceeds the scope of this paper. Providing such detail is an excellent opportunity for further research and contribution.

Systems Operation

System operation and attack patterns are related in two ways. First, attack patterns can guide design of secure operational configurations and procedures. Second, operational knowledge of security issues observed in the fielded system can be used to feed back into the attack pattern generation process.

In many cases, software with known vulnerabilities may be deployed because it may be too expensive to fix the problems, no other alternatives may be available, or it may be less expensive to design operational configurations and procedures to react to attacks instead of actually mitigating the issues in the software itself. Having proper operational configurations and procedures in place also is essential, even if software is highly secure. As described in the Leveraging Attack Patterns for Testing in the Operational Environment[23] section above, environmental conditions can dictate whether certain vulnerabilities are present in deployed software, and a large part of environmental conditions consist of operational configurations and procedures. Hence, proper operational configurations and procedures are essential to creating a secure environment [Graff 03[24]].

Attack patterns describe how an attacker may actually exploit software. Given an attack pattern, there may be ways in which certain operational procedures or environmental configurations can thwart the type of attack. For instance, procedures could be put in place to deal with the decompression bomb attack described in the Attack Pattern Generation[25] article until a vendor patch becomes available. Such procedures may include manually deleting or quarantining suspicious e-mails or temporarily blocking all external e-mail access to an organization.

Operations people, with their knowledge of security issues and familiarity with the methods of attackers in an operational environment, can also be a great source for generating new attack patterns. When indications that a system was successfully exploited are present, an investigation that identifies how the attack was carried out is generally conducted. The process described in the Attack Pattern Generation[26] article can be used during the investigation to potentially generate new attack patterns. These new attack patterns can then be leveraged to modify existing software and/or environmental configurations or to create additional operational procedures for added security.

Policy and Standard Generation

While, as described in the above sections, attack patterns can certainly be used directly by designers and developers, it is also helpful in many organizations to use attack patterns indirectly during the SDLC by

23. #dsy588-BSI_leveraging
24. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html#dsy587-BSI_graff03 (Attack Pattern References)
25. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/586-BSI.html (Attack Pattern Generation)
26. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/586-BSI.html (Attack Pattern Generation)

using them to generate policies and standards that are in turn used to develop secure software. These policies and standards can include those generated by third parties, such as with the Payment Card Industry (PCI) standards or those generated for internal use within an organization. Using attack patterns to generate policies is mostly helpful for organizations that need to dictate security standards for other organizations (e.g., credit card consortia, government agencies) and for large software development organizations.

Using policies and standards during the SDLC is not a substitute for the knowledge of attack patterns, and organizations should not rely solely on their software development staff using policies to develop secure software for several reasons. First, it is much easier to concisely describe how software can be abused than to describe how secure software should be built. Mitigations for attack patterns also vary by the technology used. Second, waiting until policies and standards are updated using information from the latest attack patterns adds another layer of indirection that increases the amount of time it takes for software developers to implement countermeasures against the latest attack patterns.

Even though appropriate policies and standards are not substitutes for attack patterns, they are extremely helpful for day-to-day software design and development activities. Policies describe high-level rules that are applicable across all software deployed in an organization. For instance, a policy may state “all data obtained from a network must be sanitized before they are processed by any business logic.” This policy may be designed to address attack patterns such as “command delimiters” and “XSS in HTTP headers.”

Standards are refinements, often seeded with useful examles, of policies that apply to specific software and/or technologies. For example, addressing the attack pattern of “XSS in HTTP headers” in Java Servlets may require use of standards such as “all data obtained from the network that contain characters must have the following characters removed as soon they are seen by a server: ‘<’, ‘>’, ‘(‘, ‘)’, ‘;’ ”. However, it is important to note that standards should always be developed and deployed in a balanced and comprehensive fashion and not in isolation. For instance, the example in the previous sentence applied in isolation could leave a system susceptible to alternate encoding issues and thus should be coordinated and buttressed with other relevant standards as an effective package.

Policies and standards are useful in large organizations because they ensure that mitigations for attack patterns are applied uniformly across all code. Like attack patterns, policies and standards can be used in all phases of the SDLC. Policies and standards also help organizations specify minimal security controls that must be in place for other organizations that handle certain types of data.

Further Reading

The remaining support articles provide a detailed glossary[28] of terms used in this series, a detailed references[29] listing, and recommendations for further exploration[30] of the attack pattern concept.

Cigital, Inc. Copyright

Copyright © Cigital, Inc. 2005-2007. Cigital retains copyrights to this material.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

For information regarding external or commercial use of copyrighted materials owned by Cigital, including information about “Fair Use,” contact Cigital at copyright@cigital.com[1].

28. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/590-BSI.html (Attack Pattern Glossary)
29. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/587-BSI.html (Attack Pattern References)
30. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack/589-BSI.html (Further Information on Attack Patterns)
1. mailto:copyright@cigital.com

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.